Some Good Practices for R

Evan Ray

5 College/Pioneer Valley R Users Group

Feb. 19, 2012

Beware of ==

- It is common to use == to test equality, but this can cause problems.

Beware of ==

- It is common to use == to test equality, but this can cause problems.
- ```
  > if(1:5 == 1) {
  +   TRUE
  + } else {
  +   FALSE
  + }
  [1] TRUE
  Warning message:
  In if (1:5 == 1) { :
    the condition has length > 1 and only the first element will
         be used
  ```
  ```

Beware of ==

- It is common to use == to test equality, but this can cause problems.
- ```
  > if(1:5 == 1) {
  +   TRUE
  + } else {
  +   FALSE
  + }
  [1] TRUE
  Warning message:
  In if (1:5 == 1) { :
    the condition has length > 1 and only the first element will
         be used
  ```
- ```
  > if(NA == 1) {
  +   TRUE
  + } else {
  +   FALSE
  + }
  Error in if (NA == 1) { : missing value where TRUE/FALSE
      needed
  ```

An alternative to ==: `identical()`

- Instead of ==, we could use the `identical()` function:

```
> if(identical(1:5, 1)) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

-

```
> if(identical(NA, 1)) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

An alternative to ==: `identical()`

- Instead of ==, we could use the `identical()` function:

```
> if(identical(1:5, 1)) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

-
```
> if(identical(NA, 1)) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

-
```
> if(identical(as.integer(1), as.real(1))) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

Another alternative to ==: `isTRUE(all.equal())`

- We could use the `all.equal()` function (but wrap it in `isTRUE()`):

```
> if(isTRUE(all.equal(1:5, 1))) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

-
```
> if(isTRUE(all.equal(NA, 1))) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] FALSE
```

-
```
> if(isTRUE(all.equal(as.integer(1), as.real(1)))) {
+   TRUE
+ } else {
+   FALSE
+ }
[1] TRUE
```

Beware of `1:N` for sequence generation

- Here's a loop that runs a loop for a number of iterations specified by the variable `n.iter`:

```
> n.iter <- 3
> for(i in 1:n.iter) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

Beware of `1:N` for sequence generation

- Here's a loop that runs a loop for a number of iterations specified by the variable `n.iter`:

```
> n.iter <- 3
> for(i in 1:n.iter) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

- What if `n.iter = 0`? We probably want to execute the loop 0 times, but...

```
> n.iter <- 0
> for(i in 1:n.iter) {
+   print(i)
+ }
[1] 1
[1] 0
```

Alternatives to 1:N: seq_len and seq_along

- One alternative is seq_len:

```
> n.iter <- 3
> for(i in seq_len(n.iter)) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

-
```
> n.iter <- 0
> for(i in seq_len(n.iter)) {
+   print(i)
+ }
```

Alternatives to 1:N: seq_len and seq_along

- One alternative is seq_len:

```
> n.iter <- 3
> for(i in seq_len(n.iter)) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

-
```
> n.iter <- 0
> for(i in seq_len(n.iter)) {
+   print(i)
+ }
```

- Another option is seq_along:

```
> X <- numeric()
> for(i in seq_along(X)) {
+   print(i)
+ }
```

Indexing – default behavior

- In R, matrices, arrays, and data frames can be indexed using square brackets:

```
> ( X <- matrix(1:20, nrow=4) )
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> X[3, 4]
[1] 15
> X[, 4]
[1] 13 14 15 16
> X[, c(2, 4)]
     [,1] [,2]
[1,]    5   13
[2,]    6   14
[3,]    7   15
[4,]    8   16
```

Indexing – default behavior

- In R, matrices, arrays, and data frames can be indexed using square brackets:

```
> ( X <- matrix(1:20, nrow=4) )
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> X[3, 4]
[1] 15
> X[, 4]
[1] 13 14 15 16
> X[, c(2, 4)]
     [,1] [,2]
[1,]    5   13
[2,]    6   14
[3,]    7   15
[4,]    8   16
```

- By default, the data type you get back depends on its dimension.

Indexing – the drop=FALSE argument

- The drop=FALSE argument can be used to make sure the original data type is returned:

```
> X[, 4]
[1] 13 14 15 16
> X[, 4, drop=FALSE]
     [,1]
[1,]   13
[2,]   14
[3,]   15
[4,]   16
```

Indexing – An Example (slide 1)

- Here's function that calculates the sum of each row of a matrix, but including only the columns specified by inds

```
> subset.sums <- function(X, inds) {
+ apply(X[,inds],1,sum)
+ }
>
> subset.sums(X, c(1, 3, 4))
[1] 23 26 29 32
```

Indexing – An Example (slide 1)

- Here's function that calculates the sum of each row of a matrix, but including only the columns specified by inds

```
> subset.sums <- function(X, inds) {
+ apply(X[,inds],1,sum)
+ }
>
> subset.sums(X, c(1, 3, 4))
[1] 23 26 29 32
```

- What happens if we include only 1 column in inds?

```
> subset.sums(X, 3)
Error in apply(X[, inds], 1, sum) : dim(X) must have a
    positive length
```

Indexing – An Example (slide 1)

- Here's function that calculates the sum of each row of a matrix, but including only the columns specified by inds

```
> subset.sums <- function(X, inds) {
+ apply(X[,inds],1,sum)
+ }
>
> subset.sums(X, c(1, 3, 4))
[1] 23 26 29 32
```

- What happens if we include only 1 column in inds?

```
> subset.sums(X, 3)
Error in apply(X[, inds], 1, sum) : dim(X) must have a
    positive length
```

- The Problem: `apply` doesn't work on vectors, and when we pull only one column out of `X`, the return value is a vector.

Indexing – An Example (slide 2)

- The Solution: Use the `drop=FALSE` argument when indexing `X`.

```
> subset.sums <- function(X, inds) {
+ apply(X[,inds, drop=FALSE],1,sum)
+ }
>
> subset.sums(X, 3)
[1]  9 10 11 12
```

Thanks!

Thanks for listening!